Lab 2: Strings and lists; boiler-plate; Cryptography

This lab gives you practice working with strings and lists. It also introduces the module "boiler-plate", which is an important concept for structuring your code. The first case study is a small module to check password strength. The main case study is a module that implements encryption and decryption using the famous Caesar Cipher.

1. Python Basic Data Types – Strings

Python identifies strings as a **sequence/collection** of **characters** enclosed in quotation marks.

Python strings can be enclosed in single or double quotation marks. This can be useful if the string contains quotation marks, as you don't need to use an escape character.

```
>>> str_var1 = 'Don\'t Panic!!'          # Using an escape character
>>> str_var2 = "Don't Panic!!"           # No escape character needed
```

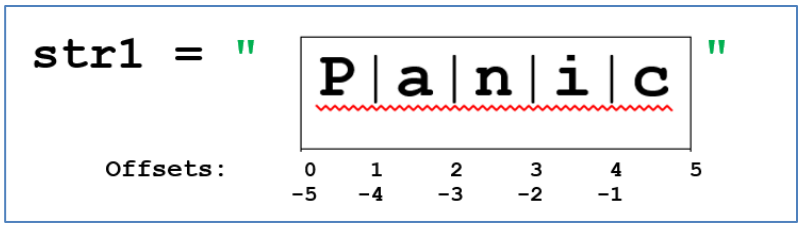
Q: When printed out, are str_var1 and str_var2 similar?

Python can perform a variety of operations on strings.
You already used the BIF len() to calculate the length of a string for the keyspace calculator script in the first lab:

```
>>> len(str_var1)
13
```

2. Slicing Strings

The slicing operator `[]` works with the string offsets/index positions to returning one or more characters from the string. The offsets for the example str1="Panic" are shown on the right:



Let's use our previously defined str_var1 to experiment with slicing.

```
>>> str_var1[1]
'o'
>>> str_var1[11]
!
```

Q: What is the index of the first character in a string?

Q: What is the value of **`str_var1[3]`**?

Python can also slice backwards from the end of the string:

```
>>> print (str_var1[-3])      # Print 3rd last char in string
c
```

Q: What is the value of **`str_var1[-7]`**?

Q: What negative index returns the last char in the string?

Q: How could the last letter be returned without negative indexing, using len() as part of the slice?

Python can also return sub-sections of a string, using slicing, using the **`[:]`** operator.

```
>>> print (str_var1[1:3]) # Slice start 1 to end 3
on
>>> print (str_var1[1:])  # Slice from start 1 to the end of string
on't Panic!!
```

Q: What slice statement returns the first 3 chars of the string?

Q: What does **`str_var1[:-2]`** extract from the string?

Q: What is the value of **`str_var1[::-1]`**?

Q: Explain what **`[::-1]`** does.

The * operator can be used to repeat the same string. It concatenates the string multiple times with itself.

```
>>> print ('ha'*4)
'hahahaha'
```

3. String Membership Operators

The **in** operator can be used to check if strings contain other strings**.**

```
>>> str_var1 = "Don't Panic!!"
>>> "Pan" in str_var1
True
>>> "pan" in str_var1     # string comparison is case sensitive
False
```

The **not in** operator can be used to check if strings do not contain other strings**.**

```
>>> "Snake" not in str_var1
True
>>> "Pan" not in str_var1
False
>>>
```

string.find() and string.index() methods

If you want the position where a substring occurs in a string, use .find() or .index().

In the lecture, we went through some .find() examples. Repeat these for practice.

```
>>> s= 'Good morning every1'
```

Try **s.index('m')**, **s.find('m')**, **s.find('z')** and **s.index('z')**, then answer the question below.

Q: What is the difference between **.find()** and **.index()**?

4. Case of Strings

Python can convert strings to upper or lower case using the built in string methods lower() and upper(). These methods return a new string; all upper or lower case (except non letters, which are copied without change).

```
>>> str_var5 = "Please don't panic! Count to 10"
>>> str_upper = str_var5.upper()
>>> str_lower = str_var5.lower()
>>> print (str_upper)
"PLEASE DON'T PANIC! COUNT TO 10"
>>> print (str_lower)
"please don't panic! count to 10"
```

Q: What built in string method can be used to create a new string `str_capital` from `str_var5` with the first character of every word capitalised?

Hint: you'll find the answer in https://docs.python.org/3/library/stdtypes.html#string-methods

Python also has a built in string method to capitalise only the very first character of a string, **`.capitalize()`.**

```
>>> print (str_var5.capitalize())
Please don't panic! count to 10
```

As Python is interpreted, string literals are converted into string objects at runtime, and so methods can be called on string literals directly:

```
>>> print ("DON'T PANIC, take 5 mins ...".capitalize())
Don't panic, take 5 mins ...
```

Python also has built in string methods to check whether strings are upper or lower case:

```
>>> "DON'T".isupper()
True

>>> "DON'T".islower()
False
```

Q: What is the Python string method to check whether a string is numeric?

Hint: you'll find the answer in https://docs.python.org/3/library/stdtypes.html#string-methods

5. Iteration of Strings

A string can be iterated through, with tasks performed on each character. A script could be written to print out each item of the string s:

```
>>> s = "Don't Panic and Code Some Python"
>>> for char in s:
        print (char)
```

Q: How many chars per line are printed?

You will see another string iteration in the next exercise.

6. String formatting: f-strings and .format()

Strings can be *joined and formatted at the same time* using f-string or .format() notations:

```
>>> home='Forfar'
>>> away='East Fife'
>>> home_score, away_score=4,5
>>> str_var5=f"Final score was {home} {home_score} {away} {away_score}."
>>> print(str_var5)
Final score was Forfar 4 East Fife 5.
>>> str_var6 = "Final score was {} {} {} {}..." .format(home, home_score,
away, away_score)
>>> print(str_var6)
Final score was Forfar 4 East Fife 5...
```

Try this out yourself!

You will see another string iteration in the next exercise.

7. Case Study: Password Strength Checker Script

Create a script in a file called **passwd_checker.py**. You can copy&paste the code from below to create the initial script, or download it from moodle.

For the purpose of this exercise, let's say a password is a strong password if it has all of the following properties:

- At least length 6;
- Contains at least one upper case character;
- Contains at least one lower case character;
- Contains at least 3 numeric characters.

As is, the code should run but will declare all passwords to be weak. Add the missing code in the function check_strength(), to check whether a password is strong. The logic of the check_strength() function is as follows:

- If the password is too short, it cannot be strong, so there is no need to check what characters it contains
- If it is long enough, we need to check every character to see if it is lower case, upper case or numeric. Keep a count of numeric characters. The variable hasupper and haslower are set to False initially. They become True when the first lower or upper case character is found, and then remain True.

Script starting point:
password_checker_start.py **(in Moodle)**

The script should run without errors, but not do too much yet!

```
# Script:  passwd_checker_start.py
# Desc:    Check strength of a password.
# Author:  Petra L, Rich Macf
# modified: Sept 2018

def check_strength(passwd):
```

```
    """checks the strength of a password"""
    # check password length
    if len(passwd) < 6:
        return False

    # check password for uppercase, lowercase and numeric chars
    hasupper = False;
    haslower = False;
    digitcount = 0;
    for c in passwd:
        #
        # add your code here
        # check whether c is upper case, lower case or numeric
        # and set corresponding variable to True or increment the counter
        #
        pass # remove this line
    if hasupper and haslower and digitcount>=3:
        return True

def main():
    # ask user input. Could use PASWord123 and 123xY to test
    # passwd = ''
    passwd = input('Enter your password to check its strength: ')
    # call strength checker function
    result = check_strength(passwd)
    # print the result
    if result: print (f'[*] Password {passwd} is strong')
    else: print(f'[*] Password {passwd} is NOT strong')

if __name__ == '__main__':
    main()
```

8. Module Boiler-plate

In the password strength checker script, you may have noticed the strange two lines at the end:

```
if __name__ == '__main__':
    main()
```

This is called a "boiler-plate". This code makes the main() function run automatically if the script itself is run, for example via IDLE. If the file is imported in another script, __name__ is not '__main__', and therefore main() is not called.

A boiler-plate in Python is a block at the end of the script. The main reason for having a boiler-plate is that it allows the same module (.py file) to be used in two different ways – it can be executed directly as a script, or be imported and used in another module or interactively. By doing the “main” check, you can have that code only execute when you want to run the module as a program and not have it execute when someone just wants to import your module and call your functions themselves.

Q: You have already **run** the script in the previous exercise using run>module or F5. What happened in this case?

Q: What is the output when you **import** the module with `>>> import passwd_checker` ?

As you have imported the module, you can now use the check_strength function:

```
>>> passwd_checker.check_strength('aB14xy67')
>>> passwd_checker.check_strength('a')
```

Q: Is the output from these commands as expected?

You should use this boiler-plate in all your scripts from now on.
Read http://stackoverflow.com/questions/419163/what-does-if-name-main-do to find out more about reasons for using boiler plate code.

9. Python Collection Data Types - Lists

Lists in Python are objects which can contain collections of other objects. They can be composed of any types of objects.

A list can be created using:

```
>>> list = [1, 'Brian ', 'Knight', 1.23,'this is a long string ']
```

Or like this:

```
>>> passwords = [    'password',
                     'qwerty',
                     'default',
                     1234,
                     12345,
                     123456]
```

A list can be a collection of floats, integers, strings and any other type of objects. This is a major difference from the static typing used for arrays in languages such as C# or Java.

Similar to Strings, lists can be sliced by specifying the offset of the data item in the list using the [] operator. (As always, offsets start from 0)

```
>>> passwords[1]
'qwerty'
```

From the >>> interpreter prompt in the IDLE Python shell window, create the passwords list shown above.

Q: What is the value of the following?

>>> passwords [0]

>>> passwords [5]

Let's have a look at some BIFs which can operate on a List.

To quickly print the list, use `print` (or just type the name of the list and hit enter):

```
>>> print(passwords)
['password', 'qwerty', 'default', 1234, 12345, 123456]
```

The length of a list is commonly needed. The `len()` BIF can be used to find this.

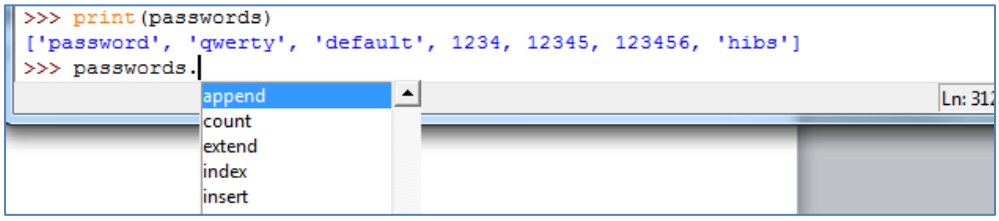
Q: What is the result of the following?

```
>>> print(len(passwords))
```

10. List Object Methods

List specific functionality is provided via *methods* on the List object itself, which use the *`Object.Method()`* syntax.

There are 3 ways to check which methods are available on an object: help() and dir() BIFs, or, in IDLE, you can just type the name of the object, a dot, and wait for the popup.



Q: List a couple of the available methods of the List object, and consider what they might do for us?

To add a single object to the end of the list, the `append()` method can be used.

```
>>> passwords.append('hibs')
>>>
>>> print passwords
['password', 'qwerty', 'default', 1234, 12345, 123456, 'hibs']
>>>
```

Add another commonly used password, such as a local sports team or a pet's name to your list.

Q: What is the difference between this and adding a character to a string?

(try it and compare the results!)

The `pop()` method can be used to remove an item from a list. The method returns the removed item.

```
>>> passwords.pop()
```

Print the list to check which items remain.

Q: Which item (offset index) did the pop() method remove from the List object?

The built in Python `pop()` method can also be used to remove an item from the middle of a list, using the item's offset.

```
>>> passwords.pop(1)
```

Print the list to check which items remain.

Q: Which item did the pop() method remove from the List object now?

Single objects in a list can be removed **by name** using the `remove()` method:

```
>>> print(passwords)
['password', 'qwerty', 'default', 1234, 12345, 123456, 'hibs']
>>>
>>> passwords.remove(123456)
>>>
>>> print(passwords)
['password', 'qwerty', 'default', 1234, 12345, 'hibs']
```

To add single objects in the middle of a list, the `insert()` method can be used, again by offset.

```
>>> passwords.insert(3,'python')
>>> print(passwords)
['password', 'qwerty', 'default', 'python', 1234, 12345, 'hibs']
```

Q: How would you use the insert() method to add 'password123' to the list after password?? Q: How would you use the insert() method to add 'qwerty123' to the list after qwerty?

To add multiple objects at the end of the list, the `extend()` method can also be used.

```
>>> passwords.extend([123456, 1234567])
>>> print(passwords)
['password', 'default', 1234, 12345, 123456, 1234567]
>>>
```

Q: What does the extend() method take as input?

Q: How would you use the extend() method to add two more numeric passwords, 12345678 and 123456789?

The list object method **sort** can be used to order the list in various ways.

In this case our passwords list contains both numbers and strings, so we need to specify how we want to sort. Otherwise we get an error message as seen in the screenshot below. To sort alphabetically, say key=str. Or, if your list contains only numbers and strings that contain only digits, you can sort numerically with key=int.

```
>>> passwords = [123456789,12345678,1234567,123456,12345,1234,'hibs','qw
erty','default','password']
>>> passwords.sort()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    passwords.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
>>> passwords.sort(key=str)
>>> print(passwords)
[1234, 12345, 123456, 1234567, 12345678, 123456789, 'default', 'hibs', '
password', 'qwerty']
```

The .reverse() method can be used to reverse a list. Both .sort() and .reverse() CHANGE the existing list forever.
If you don't want to change the list itself but see how it would look when sorted, use the **sorted()** function instead (see lecture slides). Another benefit is that sorted() can be used with any iterable.

11. Accessing Objects in a List Object – Indexing/Slicing

Any element from the list can be accessed using the offset, however, if the element does not exist then an IndexError exception is thrown:

```
>>> passwords[0]
'password'
>>>
>>> passwords[-1]
'hibs'
>>>
>>> passwords[99]

Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    passwords[99]
IndexError: list index out of range
```

Q: Which BIF would you use to check the index upper limit, to prevent an error?

```
>>> passwords[len(passwords)-1]
'hibs'
>>>
```

Similar to strings, Python can return sections of a list, using *slicing*, with the `[:]` operator.

Q: What does the following return?

```
>>> print(passwords[0:5])
```

Q: Is the original list changed? Yes/no

Q: Why/Why not?

Q: Is this the same as how the slice operates on string objects? (test this before you guess)

```
>>> print(passwords[0:5])
[1234, 12345, 123456, 1234567, 12345678]
>>> print(passwords)
[1234, 12345, 123456, 1234567, 12345678, 123456789, 'default', 'hibs', 'password
', 'qwerty']
```

Similar to Python Strings, Lists can be concatenated by using the '+' operator.

```
>>> print(passwords + [4321, 54321])
[1234, 12345, 123456, 1234567, 12345678, 123456789, 'default', 'hibs', 'password
', 'qwerty', 4321, 54321]
```

Q: Is the original list changed?

A: Again, the print statement does not change the list.

```
>>> print(passwords)
[1234, 12345, 123456, 1234567, 12345678, 123456789, 'default', 'hibs', 'password
', 'qwerty']
```

12. Iteration over Lists

Often Lists need to be iterated over, with tasks performed on each item. A simple script could be written to print out each item of the passwords list:

```
passwords = [1234, 12345, 'default', 'password', 'qwerty']

print(passwords[0])
print(passwords[1])
print(passwords[2])
print(passwords[3])
```

This will work, but it is clearly a very awkward way of printing the list - it is not "future proof" because it only works if the list stays the same size. A much better way to iterate the list is therefore to use a loop, and not to repeat the print code.

```
passwords = [1234, 12345, 'default', 'password', 'qwerty']

for password_item in passwords:
    print(password_item)
```

Try this code yourself.

Q: Does the loop iterate and print each item in the list?

Note that we can iterate over the list itself, we don't need to say "from 0 to end" and we don't need to know or calculate the length. Python handles all that internally.

13. List comprehension

List comprehension is an elegant way to iterate over lists in Python. It is often used to create a new list from an existing list that follows a specific pattern. For loops on lists can be replaced by a list comprehension; the list comprehension is usually more efficient and more elegant.

Example creating a list with a list comprehension:

>>> numbers = [n for n in range(10)]

Q: What does numbers contain?

Now let's say we want a new list, doubled_numbers that contains each of these numbers doubled.

The for loop below	Could be replaced with
`doubled = []` `for n in numbers:`	`doubled = [n*2 for n in numbers]`

`    doubled.append(n * 2)`	

Try it! A list comprehension can also be conditional, i.e. it can use an if statement to only operate on selected list items. For example:

```
>>> numbers2 = [n/2 for n in numbers if n % 2 == 0]
```

Try this code and run it.

Q: What does numbers2 contain?

Q: What is len(numbers2)?

Q: Explain exactly how this code works, in your own words:

Q: How could you achieve the same result using a for loop?

14. Nested lists

A list can contain any type of object. Therefore, a list can contain other lists, or *nested lists*. Let's extend on our passwords list, to include some alternative passwords in nested lists:

```
passwords = [1234,
             12345,
             'default', ['default1', 'default12', 'default123'],
             'password',
             'qwerty'
             ]

for password_item in passwords:
    print(password_item)
```

Try the code above - save it as a script passwords.py and run it.

Q: How is the nested list printed?

Q: What does the following return? Why? `>>> print(passwords[3][2])`

15. Iteration of nested Lists

To iterate over all elements in the main list and in the nested sublists, we can use a conditional **`if`** statement, along with the BIF **`isinstance()`** to check if the item is a list or not. You saw some examples of isinstance() in the lecture. Revise these if you cannot remember.

For example:

```
>>> if isinstance(l, list):
        print("It's a list")


It's a list
>>>
```

Q: How can you extend the passwords script to print each individual item in the list, including the nested list, giving the output shown below?

Try it out and save your code.

```
>>> ============================== RESTART ==============================
>>>
1234
12345
default
default1
default12
default123
password
qwerty
>>>
```

16. More String formatting and flattening lists with .join()

Now that we have practiced using both strings and lists, let's have a look at the .join() method. This is a bit strange at first, but can be really useful for joining strings and flattening lists. It's probably easiest to explain with an example:

```
>>> str_var1="Don't Panic!"
>>> list_var1 = [str_var1, "and", "carry", "on"]
>>> " ".join(list_var1)
"Don't Panic! and carry on"
```

In this example, the .join() method is applied to the string " " (a single space character). The argument of the method is the list list_var. As you can see, this joins all the elements of the list into a single string using " " as the separation character. In effect, we have "flattened" the list.

The separation character can be a string of any length:

```
>>> list_var2 = ["Lions", "Tigers", "Bears"]
>>> " and ".join(list_var)
'Lions and Tigers and Bears'
```

Q: What are the results of the two joins below? Can you explain what is happening here?

```
>>> "?".join(str_var1)


>>> str_var1.join('???')
```

17. Python Basic Data Types – Scientific notation for numbers

Last week, with the keyspace.py script, you saw some output in scientific notation. For example, you may have found that there are 6.302494097246094e+17 (math.pow(95,9)) possible ascii passwords with 9 characters.

Q: How can you use python to find out what 6.302494097246094e+17 is when written as a standard integer, without scientific notation? (there are several methods – how many can you think of? – see lecture slides)

Q: The chances of randomly guessing a 3-character ascii password at the first attempt are p=1/(95^3). When Python displays the result of this, in scientific notation, what is the e-value? How many zeroes would appear after the decimal point when this is written out in full?

Q: Use a print statement with f-string notation to display these chances as a float with 7 decimal places (**.7f**). [refer to the lecture slides and/or later exercises below!]

The screenshot below shows how I used the .format() method. Can you work out what is greyed out here?

```
>>> out = 'The chances of randomly guessing a 4 character password \
at the first attempt are {[redacted]} or {[redacted]}'.format([redacted])
>>> print(out)
The chances of randomly guessing a 4 character password at the first att
empt are 0.00000001 or 1.2277376631548254e-08
```

18. Case Study: Cryptography with the Caesar Cipher

Create a module called **caesar.py**. You can copy&paste the code from the link below to create the initial script or download it from moodle.

The script should run without errors, but not do too much yet!

Script starting point:
caesar_start.py **(in Moodle)**

```
# Script:  caesar.py
# Desc:    encrypt and decrypt text with a Caesar cipher
#          using defined character set with index
# Author:  your name. Based on a template
# Created: 23/9/17

charset="ABCDEFGHIJKLMNOPQRSTUVWXYZ" # characters to be encrypted
numchars=len(charset) # number of characters, for wrapping round

def caesar_encrypt(plaintext,key):
    """put an appropriate function doc string here"""
    print (f'[*] ENCRYPTING - key: {key}; plaintext: {plaintext}')

    # plaintext=     # convert plaintext to upper case
    ciphertext=''    # initialise ciphertext as empty string

    for ch in plaintext:
        if ch in charset:
            new='' # replace this with your code, may use extra lines
        else:
            new=ch # do nothing with characters not in charset
        ciphertext=ciphertext+new
    print (f'[*] ciphertext: {ciphertext}')
    return ciphertext # returns ciphertext so it can be reused

def caesar_decrypt(ciphertext,key):
    """put an appropriate function doc string here"""
    # very similar to caesar_encrypt(), but shift left
    print (f'[*] DECRYPTING - key: {key}; ciphertext: {ciphertext}')
    #
    plaintext=''   # replace this with your code
    #
    print (f'[*] plaintext: {plaintext}')
    return plaintext # returns plaintext so it can be reused

def caesar_crack(ciphertext):
    """put an appropriate function doc string here"""
    # how could you brute force crack a caesar cipher?
    # your code here
```

```
def main():
    # test cases
    key=2
    plain1 = 'Hello Suzanne'
    cipher1 = 'IQQfOQtpKpIGXGtaQPG'
    crackme = 'PBATENGHYNGVBAFLBHUNIRPENPXRQGURPBQRNAQGURFUVSGJNFGUVEGRRA'
    # call functions with text cases
    caesar_encrypt(plain1, key)
    caesar_decrypt(cipher1,key)
    # caesar_crack(crackme)  # remove comment to test cracking

# boilerplate
if __name__ == '__main__':
    main()
```

Your task is to complete the script appropriately so that the functions work as expected. Use the lecture notes to help you get started.

Ideally, any spaces should also be removed from the string during encryption.

19. Optional Challenge exercise - nested list comprehension

In Exercise 15 above, you used a loop with an if statement to print every element of a nested list. Rewrite this as a nested list comprehension.

Two references you may find useful for this are http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/ and https://campus.datacamp.com/courses/python-data-science-toolbox-part-2/list-comprehensions-and-generators?ex=5.

20. References

Python list comprehensions: Explained visually. http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/

Python 3.7 Documentation. (n.d.), https://docs.python.org/3/tutorial/datastructures.html; (up to 5.2, the del statement)

NakedSecurity (2010). The top 50 passwords you should never use. https://nakedsecurity.sophos.com/2010/12/15/the-top-50-passwords-you-should-never-use/

Read http://stackoverflow.com/questions/419163/what-does-if-name-main-do to find out more about reasons for using boiler plate code.

How to use extended slices https://docs.python.org/2/whatsnew/2.3.html#extended-slices

f-strings are a new feature introduced for Python 3.6. Find out more by reading Literal String Formatting (f-strings) at https://docs.python.org/3/whatsnew/3.6.html and https://docs.python.org/3/reference/lexical_analysis.html#f-strings.